The Thesis Committee for Zakaria Alrmaih
Certifies that this is the approved version of the following thesis:

Korat-API: An API for building constraint solving problems for Korat

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Sarfraz Khurshid

Lingming Zhang

Korat-API: An API for building constraint solving problems for Korat

by

Zakaria Alrmaih, B.S.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2017

Dedication

Dedicated to my beloved parents and brothers.

Acknowledgements

I wish to express my appreciation to my supervisor Dr. Sarfraz Khurshid, for his continuous support and patience, motivation, and immense knowledge. Further, I would like to thank Dr. Lingming Zhang for taking the time to review this work.

Nima Dini and Cagdas Yelen kindly helped in developing the foundational ideas in this thesis.

Hayes Converse and Corina Pasareanu kindly helped with using the Symbolic PathFinder tool.

This work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-1319688 and CNS-1239498

Abstract

Korat-API: An API for building constraint solving problems for Korat

Zakaria Alrmaih, M.S.E

The University of Texas at Austin, 2017

Supervisor: Sarfraz Khurshid

This thesis introduces the foundation of an API for building constraint solving problems for the Korat solver for imperative predicates. Our goal is two-fold: (1) to facilitate the use of Korat as a backend solver for applications that desire using it as a constraint solving engine; and (2) to facilitate optimized analyses using Korat, which follow the spirit of modern constraint solving and software testing techniques. We describe the API and how it uses the core Korat engine, and demonstrate the benefits in two application contexts: (1) using Korat as a backend engine for model counting; and (2) using Korat as test generator. We believe our work introduces a promising approach for making the ability of Korat to efficiently solve imperative predicates more widely applicable, possibly even in new application contexts where Korat has net been used before.

Table of Contents

List of Figures

Chapter 1: Introduction

Constraint solvers [1]-[3] are playing an increasingly important role in the development of robust and reliable systems. For example, efficient constraint solving lies at the heart of various modern approaches to symbolic execution [4]-[6], test input generation [3], [7], and automated theorem proving [8].

The focus of this thesis is the Korat constraint solver [3], which introduced the idea of solving constraints written as imperative predicates, termed *repOk()* methods [9], in Java for automated constraint-based test case generation. Specifically, the Korat user describes properties of desired test inputs by writing the boolean *repOk()* method that checks those properties for its given input and returns true if and only if the input satisfies them. In addition, the users write a *finitization* that bounds the input size, using also Java code.

Korat implements a backtracking search with efficient pruning and isomorphism breaking to systematically explore the space of all possible inputs up to the finitization bound, and enumerates all inputs that satisfy the desired properties; each generated input serves as a desired input for testing the program under test. Korat's pruning is *execution-driven* where it repeatedly executes *repOk()* on candidate inputs, observes the executions, and creates new candidates based on object fields accessed during the execution.

The basic Korat approach has been applied in various applications for automated testing [10]; most recently, it was used as a backend tool for model counting [11], [12]. Moreover, several projects have built on the basic Korat to further optimize it, for

example using parallel techniques [13]-[15], incremental techniques [15], memorization [15], and static analysis [16].

A key property of Korat that makes using it particularly attractive for automated constraint-based test generation is that it solves predicates written in imperative code and thus doesn't require the use of some specialized language for writing constraints, which has semantics and syntax possibly very different from widely used imperative programming languages that the developers are commonly familiar with. Thus, Korat users who already know Java can continue to use it to automate testing of their program without having to learn an unfamiliar constraint language or programming paradigm.

While this key property is a particular strength of Korat, it also leads to a basic limitation of Korat: the constraints that are logical in nature must be written in a language that is *imperative*, which can create a potential mismatch between "what was intended" and "how it was written". The logical nature of the constraints can largely be lost due to the required use of standard imperative constructs, e.g., short-circuiting of conditional expressions. For example, assume the user wants the inputs to satisfy properties *p* and *q* that can each be checked independently, i.e., regardless of whether the other property holds, and writes the *repOk()* predicate body simply as "*return p() and q();"* where predicate *p()* checks property *p* and predicate *q()* checks property *q*. Purely as a consequence of the way the user has written the *repOk(),* an artificial dependence, specifically *q* is checked only if *p* is true, i.e., *p()* has become a pre-condition of *q()*, between the two properties has been introduced.

Preserving the logical structure of the constraints is valuable for two key reasons. One, it facilitates re-use of constraints written as imperative predicates and understanding them. Two, perhaps more importantly, it allows the backend search to be improved using optimization techniques which are able to explore different orders of checking properties,

e.g., as envisioned in previous work [17]. The execution-driven nature of Korat's backtracking search makes it inherently dependent on the way the *repOk()* is written. The same properties written differently can lead to significantly different performance of Korat. Thus, for optimal performance, the user has to pay particular attention to how to write *repOk()*.

Our thesis is that we enhance the applicability and usefulness of Korat by defining an API that allows building constraint solving problems for Korat and enables directly conveying (1) high level structure for the desired input properties being written as *repOk()* predicates, and (2) specific search goals, e.g., based on test criteria. We design our API specifically in view of some projects on parallel technique that enhance Korat [13]-[15] or use it as backend solver [11], [12].

This document describes the foundation of our API, which we term Korat-API, and how it uses the core Korat engine, and demonstrate its benefits in two application contexts – as a backend constraint solver; and as a standalone test generator. In the context of using Korat as a backend constraint solver, we focus on a recent approach for reliability that leveraged the standard Korat tool-set and invoked directly [11], [12]. Specifically, we demonstrate how Korat-API provides the support necessary to construct constraint solving problems for model counting when the predicates are based on path conditions from symbolic execution [4]. In the context of using Korat as standalone test generator, we focus on two way to enhance Korat. Specifically, we demonstrate that our API supports describing (1) *logical* conjunctions of properties, e.g., to support approaches that re-order constraints for faster solving [17], (2) *separable* properties, i.e., constraints that are on disjoint parts of the input, e.g., to support parallel approaches.

We believe our work introduces a promising approach for making the ability of Korat to efficiently solve imperative predicates more widely applicable, possibly even in

new application contexts where Korat has not been used before. Our work takes inspiration from Kodkod backend [19] for Alloy [20], a first-order declarative language with transitive closure, which is particularly suitable for creating software models. Kodkod provides a Java API that allows other application to create constraint solving problems in Alloy and solved using off-the shelf propositional satisfiability (SAT) solvers. Thus, the Kodkod API provides a programmatic interface for creating constraints in first order logic and solving them using SAT. Kodkod has enabled the Alloy tool-set and SAT solvers to be utilized in various applications and tools [21]. We hope our work open the possibility of utilizing Korat's search capabilities likewise.

This thesis makes the following contributions:

- **Korat API**: We introduce the idea of providing an API for constructing constraint solving problems for the Korat solver for imperative predicates.
- **Translation to imperative solving problems**: We describe the core algorithms that allow our API to be used for constructing solving problem based on imperative constraints.
- **Demonstration**: We present a two-fold demonstration of using our API. One, we use it to show how an application may use Korat as backend solver for model counting. Two, we show how it allows using the standard Korat algorithm for input generation.

Chapter 2: Illustrative Examples

This section presents two examples to illustrate Korat-API.

2.1 MODEL COUNTING AND SEPARABLE CONSTRAINTS

Consider using Korat as a backend model counter to count the number of solutions for a constraint derived from path conditions that arise in symbolic execution, e.g., in the spirit of recent work on software reliability [11], [12]. To illustrate, consider the following path condition:

header != null && header.next != null && header.next.next == header && size == 2

where *header* and *size* are fields in class List and *next* is a field in class *Node*.

Figure 1 shows example Java code that the user can write to create the constraint solving problem that represents this path condition. The body of the *repOk()* method returns true if and only if the path condition holds.

The following Korat invocation solves this problem and reports the count of the solutions: "*java -cp Korat.jar korat.Korat --class korat.examples.list.List --args 0,3,3"*
Executing this command returns output shown in Figure 2.

```
import java.util.*;
import korat.finitization.*;
import korat.finitization.impl.*;


public class List {
    Entry header;
    int size;
    public static class Entry{
      Entry next;
    }
    public boolean repOK() {
      boolean result = false;
      try {
          result = header != null && header.next != null && header.next.next == header && size == 2;
      } catch (Exception e) {

      }
      return result;
    }
    public static IFinitization finList(int min, int max, int num) {
      IFinitization f = FinitizationFactory.create(List.class);
      IIntSet sizeRange = f.createIntSet(min,max);
      f.set("List.size", sizeRange);
      IObjSet entries = f.createObjSet(Entry.class,true);
      entries.addClassDomain(f.createClassDomain(Entry.class,num));
      f.set("List.header", entries);
      f.set("Entry.next", entries);
      return f;
    }
}
```

Figure 1: Creating a constraint solving problem for counting the number of solutions for path condition "header != null && header.next != null && header.next.next == header && size == 2" using standard korat [3]

Figure 3 shows how Korat-API can be used to define a constraint solving problem that is analogous to the problem represented in Figure 1 and its corresponding Korat invocation above. Lines 5-7 list the imports needed to run the problem. Lines 9-11 declare the object fields. Lines 13-28 define the *repOk()* predicate. Note the invocation of *sand()* to create a separable formula that is a conjunction of two constraints that can be solved in parallel. Lines 33 and 35 define the overall value domain. Lines 34, 36, and 37 define for each field, its value domain. Lines 39-45 build the constraint solving problem and invoke the default Korat solver to compute the count of the solutions as well as print it.

```
Start of Korat Execution for List (repOK, [0, 3, 3])

[0]List@c2f8b5a.size -> int:4
[1]List@c2f8b5a.header -> class List$Entry:4
[2]List$Entry@565c7f6.next -> class List$Entry:4
[3]List$Entry@5d2aea3e.next -> class List$Entry:4
[4]List$Entry@2c78bc3b.next -> class List$Entry:4

0 0 0 0 0  :: 1
0 1 0 0 0  :: 1 2
0 1 1 0 0  :: 1 2 0
1 1 1 0 0  :: 1 2 0
2 1 1 0 0  :: 1 2 0 ***
3 1 1 0 0  :: 1 2 0
0 1 2 0 0  :: 1 2 3
0 1 2 1 0  :: 1 2 3 0
1 1 2 1 0  :: 1 2 3 0
2 1 2 1 0  :: 1 2 3 0 ***
3 1 2 1 0  :: 1 2 3 0
0 1 2 2 0  :: 1 2 3
0 1 2 3 0  :: 1 2 3
Total explored:13
New found:2
```

Figure 2: Korat output for counting solutions

2.2 INPUT GENERATION

Consider using Korat to generate binary trees with parent pointers as test inputs. Figure 4 shows the Java code that defines the constraint solving problem for generating binary trees with parent pointers using standard Korat [3]. The class *Node* introduces the type necessary to model the tree structure. Each tree has a *root* node. Each node has a *left* child node and a *right* child node. The predicate *repOk()* uses one helper method to check the desired properties.

```
import Design.*;
public class RunList {
    public static void main(String[] args) {
        /* ----------------------------- Imports ----------------------------- */
        String [] imports = {"java.util.*",
                             "korat.finitization.*",
                             "korat.finitization.impl.*"};
        /* ---------------------------- Declare Variables ---------------------------- */
        Var header = new Var("header","List", "Entry");
        Var size = new Var("size","List", "int");
        Var next = new Var("next", "Entry", "Entry");
        /* ---------------------------- repOk ---------------------------- */
        Expr headerExpr = new Expr(header);
        Expr sizeExpr = new Expr(size);
        Expr nextExpr = new Expr(next);
        Expr Expr1 = headerExpr.oper(nextExpr, ExprOperator.Join);
        Expr Expr2 = Expr1.oper(nextExpr, ExprOperator.Join);
        Clause c1 = new Clause(headerExpr, Op.NEQ, Expr.nullExpr);
        Clause c2 = new Clause(Expr1, Op.NEQ, Expr.nullExpr);
        Clause c3 = new Clause(Expr2, Op.EQ, headerExpr);
        Clause c4 = new Clause(sizeExpr, Op.EQ, "2");
        Formula f1 = new Formula(c1);
        Formula f2 = new Formula(c2);
        Formula f3 = new Formula(c3);
        Formula f4 = new Formula(c4);
        Formula f5 = f1.And(f2);
        Formula f6 = f5.And(f3);
        Formula fFinal = f6.sAnd(f4);
        /* ---------------------------- Finitization ---------------------------- */
        String mainClass = "List";
        String [] finArgs = { "int min", "int max" , "int num" };
        Fin finitization = new Fin(mainClass, finArgs);
        finitization.CreateIntRange("sizeRange", "min", "max");
        finitization.SetRange(size, "sizeRange");
        finitization.CreateObjRange("entries","Entry", true, "num");
        finitization.SetRange(header, "entries");
        finitization.SetRange(next, "entries");
        /* ---------------------------- Build All ---------------------------- */
        Builder builder =  Builder.GetBuilder();
        builder.mainClass("List", fFinal, finitization);
        builder.build(imports);
        /* ---------------------------- Compile ---------------------------- */
        CompilerKoratX.compile(builder.getSourceCode(), mainClass);
        /* ---------------------------- Run Korat ---------------------------- */
        RunKoratx.run(mainClass, "0,3,3");
    }
}
```

Figure 3: Constraint solving problem defined by Korat-API that is analogous to the problem represented in Figure 1

The method isAcyclic() checks if the input has a valid tree structure, i.e. has no (undirected) cycles. The finitization method specifies *root*, *left*, and *right* fields are either null or point to one of numNodes unique nodes that Korat search will create upon initialization.

```
public class BinaryTree {
    Node root;
    public boolean repOK() {
      if(!isAcyclic())
          return false;
      return true;
    }

    @SuppressWarnings({ "rawtypes", "unchecked" })
    private boolean isAcyclic() {
        Set visited = new HashSet();
        visited.add(root);
        LinkedList workList = new LinkedList();
        workList.add(root);
        while (!workList.isEmpty()) {
            Node current = (Node) workList.removeFirst();
            if (current == null)
                return false;
            if (current.left != null) {
                // checks that the tree has no cycle
                if (!visited.add(current.left))
                    return false;
                workList.add(current.left);
            }
            if (current.right != null) {
                // checks that the tree has no cycle
                if (!visited.add(current.right))
                    return false;
                workList.add(current.right);
            }
        }
        return true;
    }
    public static IFinitization finBinaryTree(int numNodes) {
      IFinitization f = FinitizationFactory.create(BinaryTree.class);
      IObjSet nodes = f.createObjSet(Node.class,true);
      nodes.addClassDomain(f.createClassDomain(Node.class,numNodes));
      f.set("BinaryTree.root", nodes);
      f.set("Node.left", nodes);
      f.set("Node.right", nodes);
      return f;
    }
}
```

Figure 4: Java code show the constraint solving problem for generating binary trees using standard Korat [3][http://korat.sourceforge.net/]

To run Korat, the user compiles the Java code that describes solving problem (Figure 4) and invokes Korat using the command line. To illustrate, "*java -cp Korat.jar korat.Korat --class korat.examples.searchtree.BinaryTree --args 2"* instructs Korat to enumerate all trees with up to 2 nodes.

Figure 5 shows how Korat-API can be used to define a constraint solving problem that is analogous to the problem represented by Figure 4 and its corresponding korat invocation above. Lines 32-34 list the imports needed to run the problem. Lines 36 declare the object field. Lines 38-39 define the *repOk()* predicate. Lines 44 defines the overall value domain. Lines 45-47 define for each field, its value domain. Lines 50-58 build the constraint solving problem and invoke the default Korat solver to compute the count of the solutions as well as print it. The helper method was provided as string to the predicate call.

```
30    public static void main(String[] args) {
31        /* ---------------------------- Imports ---------------------------- */
32        String [] imports = {"java.util.*",
33                             "korat.finitization.*",
34                             "korat.finitization.impl.*"};
35        /* ---------------------------- Declare Variables ---------------------------- */
36        Var root = new Var("root","BinaryTree", "Node");
37        /* ---------------------------- repOk ---------------------------- */
38        Predicate isAcyclic = new Predicate("isAcyclic()",isAcyclicMethod);
39        Formula fFinal = isAcyclic.Invocation();
40        /* ---------------------------- Finitization ---------------------------- */
41        String mainClass = "BinaryTree";
42        String [] finArgs = { "int numNodes" };
43        Fin finitization = new Fin(mainClass, finArgs);
44        finitization.CreateObjRange("nodes","Node", true, "numNodes");
45        finitization.SetRange(root, "nodes");
46        finitization.SetRange("Node.left", "nodes");
47        finitization.SetRange("Node.right", "nodes");
48
49        /* ---------------------------- Build All ---------------------------- */
50        Builder builder =  Builder.GetBuilder();
51        builder.mainClass(mainClass, fFinal, finitization);
52        builder.build(imports);
53
54        /* ---------------------------- Compile ---------------------------- */
55        CompilerKoratX.compile(builder.getSourceCode(), mainClass);
56
57        /* ---------------------------- Run Korat ---------------------------- */
58        RunKoratx.run(mainClass, "2");
59    }
60 }
```

Figure 5: Creating a constraint solving problem for generating binary tree using Korat-API

Chapter 3: Korat-API

This section describes our design of Korat-API, which has two key elements: (1) an API for defining constraint solving problems; and (2) algorithms for translating problems defined using the API to constraint solving problems for standard Korat. The usage of Korat-API requires four core steps:

1. Define the constraint solving problem p_a *using the API*
2. Translate the problem p_a to a constraint solving problem for p_k for standard Korat
3. Run standard Korat to Solve the problem p_k
4. Report the solution to problem p_a

Step 1 may be performed by a human user or a Java application. Steps 2, 3, and 4 are mechanical and defined by Korat-API.

3.1 API

Our API supports the following four key parts of building a constraint solving problem:

- Defining fields
- Defining Constraints
- Defining the bounds on the state space
- Defining the analysis to perform

Each constraint solving problem is defined in a new Java class that contains a main method which invokes the code methods to declare the four parts of the problem

described above. Figure 6 describes the basic structure for building constraints using the Korat-API classes.

Var := Identifier

ExprOperator := . | + | - | * | ++ | &

Expr := Var | Expr ExprOperator Expr

Op := != | == | < | > | <= | >=

Clause := Expr Op Expr

LogicalConj := And | lAnd | sAnd | Or

Formula := Predicate | Clause | Formula LogicalConj Formula

Figure 6: Abstract Syntax

3.1.1 Defining fields

Variable (Var) data structure describes the different fields used to construct the constraint solving problem. It contains the variable information like its type, name, and what class contains it. The format of the variable declaration is:

Var(String memberName, String parentClassName, String memberType)

For example, the following declaration of field *root* in *Tree* class with type *Node* is declared as following:

Var rootVar = new Var("root", "Tree", "Node");

In this example, the variable *rootVar* contains the information regarding the *root* field with its type *Node* and its parent class *Tree*.

3.1.2 Defining Constraints

Expression structure describes fields as well as the relational and arithmetic operations between them using expression operators (described below). This class describes the relation between different structures that are required to develop the *repOk()* predicate. For example, *root.right.elem* is an expression. *root* is of type *Node*. Type *Node* contains node *right*, node *left*, and Integer element *elem*. The declaration of such expressions is as following:

```
Expr rootExpr = new Expr(root);
Expr rightExpr = new Expr(right);
Expr elemExpr = new Expr(elem);
Expr Expr1 = rootExpr.oper (rightExpr, ExprOperator.Join);
Expr finalExpr = Expr1.oper (elemExpr, ExprOperator.Join);
```

We describe each field by an expression. Then, we use expression operators to describe the relation between them. *Join* is one of the expression operators that joins two fields as shown above.

As described above, expression operators are used to define the type of the relational and arithmetic operation between fields. There are several expression operators. Some of them are relational operators that describe relation between two structures. The rest are arithmetic operators that describe the math operation performed between two fields. The following table show some examples of expression operators:

Join (.)	Relational join (.) operator
Multiply(*)	Arithmetic Multiply (*) operator
Add (+)	Arithmetic Addition (+) operator

For example, if we want to represent the following expression $x+z*y$, we define it as following:

Expr xExpr = new Expr("x");

Expr yExpr = new Expr("y");

Expr zExpr = new Expr("z");

Expr Expr1 = zExpr.oper(yExpr, ExprOperator.Multiply);

Expr finalExpr = xExpr.oper(Expr1, ExprOperator.Add);

Clause represents the relation between expressions. Basically, clause provides a way to build the structural constraint to create the repOk formula. The relational operations are defined in Operation data type (described below). The evaluation of each clause returns a boolean. For example, *root != null* is a clause that evaluates to false when root is null and true otherwise. Assume *elemClause* is defined as *root.elem <= root.right.elem*, then *elemClause* can be declared as following:

Clause elemClause = new Clause(lhs, Op.LTE, rhs);

lhs and *rhs* are expressions that represent *root.elem* and *root.right.elem* respectively.

Operation (Op) type represents the type of comparison operation to preform between expressions in a clause. The operations can be one of the following: ==, != , <, >, <=, or >=. As shown in *elemClause,* Op.LTE was used to describe the type of comparison operation between two expressions.

Formula represents the constraint for the constraint solving problem. *repOk()* predicate is basically described by the formula data structure. Formula can be represented by Boolean, Clause or Predicate (described below). Formula supports Korat-API for building conjunctions of formulas to describe sequential (And), logical (lAnd), and separable (sAnd) conjunctions. For example, assume that *clause1* is $x > y$, *clause2* is $y == z$, and *clause3* is $x < z*y$. To build the formula f6 to be equal to $(x > y \&\& y == z) \mid\mid !(x < z*y)$, the following steps are performed:

```
Formula f1 = new Formula(clause1);
Formula f2 = new Formula(clause2);
Formula f3 = new Formula(clause3);
Formula f4 = f1.And(f2);
Formula f5 = f3.Not();
Formula f6 = f4.Or(f5);
```

Korat-API also support using helper predicates to construct formulas like the example shown in Figure 5. The following statement in Korat-API allow the declaration of predicate constraints:

```
Predicate p = new Predicate (String Name,  String Method-Body);
```

The following invocation method constructs the formula f using the predicate p:

```
Formula f = p.Invocation();
```

3.1.3 Defining the bounds on the state space

In finitization (Fin) class, the input size is bounded by specifying the range of possible inputs for each field. Finitization contains two core methods: (1) method to generate the possible ranges of values that field can take, e.g., Integer range from 0 to 2; (2) method to assign those ranges to desired fields to bound the possible values for that field. First, we need to construct the finitization class with the name of the main class and finitization arguments if any like the following:

Fin finitization = new Fin(String main class, String args[]);

Then, we generate the desired range.

finitization.CreateIntRange (String range name, int min, int max);

or

finitization.CreateObjRange(String range name, String number of obj);

Finally, we assign the range to the corresponding field to bound the values it can take as follow:

finitization.SetRange(Var field name, String range name);

For example, *Node* class has an integer member called *elem*. Assume we want to bound the integer value to be from 0 to 10. We follow the next two lines of code:

finitization.CreateIntRange("elem range", 0, 10);

finitization.SetRange(elem, "elem range");

3.1.4 Defining the analysis to perform

The core data structure in Korat-API is the Builder class. Builder connects all data structure to build constraint solving problem. It basically joins pieces together to construct a well-defined constraint solving problem. The following method in class *Builder* allows user to define the specific constraint solving analysis to perform:
builder.mainClass(String main class, Formula repOk predicate , Fin finitization);

The algorithm of this builder is described in detail in the next section. We use *javacompiler* to compile the code in runtime using *CompilerKorat*. Finally, in *RunKorat*, we use the generated class from *CompilerKorat* to invoke standard Korat with the desired arguments to generate all valid input cases.

3.2 ALGORITHMS

In the previous section, we have seen how to define the constrain solving problem p_a using Korat-API. In this section, algorithms of how to translate the problem p_a to a constrain solving problem p_k for standard Korat will be covered. This section is divided into three parts. The first part will define the first algorithm in which we declare the classes and associate members of each class using the variable data structure. Second, we will explain an algorithm to build the constraint using Korat-API to match the constraint used in standard Korat. Finally, we will describe an algorithm that builds the finitization method using the Fin class to bound the state space of fields. The main component in all three algorithms is the source code which is a string that holds p_k after the translation of p_a using the algorithms described in this chapter. Source code would be built to be

compiled in *CompilerKorat*. The source code is basically the string Java version of the constraint solving problem that standard Korat invokes.

3.2.1 Declare classes

As described in the grammar section about variables, each field in our constraint solving problem is declared as variable that contains its type, name, and what class contains it. In this section, we use the following syntax to represent different inputs:

T_m : The type of the member in class

T_p: The name of class containing the member (parent class)

N : name of the member or field

Recall from the API section, the variable declaration contains T_m, T_p, and N. Using those three information, we are going to declare the classes with their associate members. To simplify the algorithm, assume **member** class holds the name of the member (N) and its type (T_m). The following data structures will be used in our algorithms :

- Member[N, T_m] : Member class that holds name and type of field
- Var[N, T_p, T_m] : Variable class
- clsToMember[T_p, Set<Member>] : Store all variable declarations which is mapping between class and its members
- VC: Set of all Var declarations calls from Korat-API
- SourceCode : It is the string version of the source code that we will compile to create Korat problem

The following algorithm describes how we declare classes in p_k using variable call from p_a:

Algorithm 1: *Declare Classes in Source Code*

Input: Set of Var API calls (VC), SourceCode, and T_{main}

1. **function** DeclareClasses(VC, SourceCode, T_{main})
2. ClsToMember ← Empty Map
3. **for all** Var[N, T_{parent}, T_{member}] ∈ VC **do**
4. Member.init(N, T_{member})
5. **if** ClsToMember.get(T_{parent}) != null
6. **then** ClsToMember.get(T_{parent}).add(Member)
7. **else** ClsToMember.add(T_{parent}, Member)
8. SourceCode.append(DeclarePublic(T_{main})) /* add "public class T_{main} {" */
9. SourceCode.append(AddMembers(ClsToMember.get(T_{main})))
10. ClsToMember.remove(T_{main})
11. **for all** $T_{parentN}$ ∈ ClsToMember.keys() **do**
12. SourceCode.append(DeclareStaticPublic($T_{parentN}$))
13. SourceCode.append(AddMembers(ClsToMember.get($T_{parentN}$)))
14. SourceCode.append(CloseClass()) /* add "}" */
15. **return** SourceCode
16. **end function**

Output: Class Declarations in Source Code

Algorithm 1 converts variable declaration calls from the API to class declarations in source code. Algorithm 1 takes set of API calls (VC) and the main class specified by the user as inputs (shown in line 1). For each of those Var calls, we check if the parent class for the field already exists in ClsToMember map (shown in line 6). If yes, we add the new member to the member set associated with that class. If no, we add new mapping to ClsToMember with the parent class as the key and the new member as its value. After the loop at line 4 terminates, ClsToMember will contain all the class declarations with their associate members. In order to declare the class in source code, we need to declare the main class first. Therefore, in line 9, we declare the main class T_{main} to be public. Then, we retrieve its members from the ClsToMember using T_{main} as the key and we declare them. Then, we remove the main class from set of classes in ClsToMember. Then, we iterate over the rest of the undeclared classes in line 12 and we declare them to be static public with their members. Finally, we end the declaration of all the classes with "}". Then, we return the generated source code with the declared classes as was planned.

The above algorithm describes how each class is declared in our source code. The user specifies which parent class is the main class that contains the repOk and finitization methods. The rest of the classes would be nested under the main class. Here is an example of how we declare the class from the variable calls in our API. In this example, assume we have the four calls of variable declaration shown in Figure 7. Also, assume that *Tree* is the main class here. Following Algorithm 1, we first build the classToMember data structure. When we process the first call here:

```
Var elem = new Var("elem", "Node", "int");
```

The classToMember would be like the following

Node	Int elem

Then, when we process the second line:

```
Var right = new Var("right", "Node", "Node");
```

The classToMember would be like the following:

Node	Int elem Node Right

After we process all the commands in Figure 7, the classToMember would be like the following:

Node	Int elem Node Right
Tree	Node root Node current

Since *Tree* is the main class here, *Tree* declaration is added to the source code as the main class as well as the declaration of its members.

```
public class Tree {
    Node root;
    Node current;
}
```

Then, *Tree* class is removed from the classToMember. We iterate over the rest of classes and add their declarations with their associated members as nested class in the main class. In this case, there is only one class left to declare which is *Node*. *Node* declaration is added to the source code under the main class as well as the declaration of its members. The generated source code is shown in Figure 8.

```
Var elem = new Var("elem", "Node", "int");
Var right = new Var("right", "Node", "Node");
Var root = new Var("root","Tree", "Node");
Var current = new Var("current","Tree", "Node");
```

Figure 7: Example of Field declarations in Korat-API

```
public class Tree {
    Node root;
    Node current;
    public class Node {
        Node right;
        int elem;
    }
}
```

Figure 8: Declared classes in constraint solving problem in standard Korat from translating field declaration calls from Korat-API

3.2.2 Generate repOk

In this algorithm, we generate the constraint predicate of the constraint solving problem for standard Korat. As described before, the formula data structure is used to specify the desired constraints. In order to construct the formula, variables, expressions, clauses have to be declared first. We have seen in API section how each of those data structures were declared and used separately. In the following algorithm, combining the different classes to generate the desired formula will be illustrated.

Algorithm 2: *Steps to Generate the desired repOk using our API*

Input: Desired constraint (C), SourceCode

1. **for all** field[N, T_{parent}, T_{member}] $\in$ C **do**:
2. Var N = new Var(field))
3. **for all** field[N, T_{parent}, T_{member}] $\in$ C **do**:
4. Expr E_n = new Expr(field))
5. **for all** E_{left}, Expression Operators(EO) and E_{right} $\in$ C **do**:
6. Expr E_m = E_{left}.oper(E_{right}, EO)
7. **for all** E_{left}, Operations(Op) and E_{right} $\in$ C **do**:
8. Clause clause $_n$ = new Clause(E_{left}, Op, E_{right})
9. **for all** clause in C_n **do:**
10. Formula f_n = new Formula(clause)
11. **for all** Predicates P_n $\in$ C **do**:
12. Formula f_n = new Formula(P_n)
13. f_{final} $\leftarrow$ Conjunction(f_n, logical operators(And, lAnd, sAnd))
14. SourceCode.append(GenerateRepOk(f_{final}))

Output: Add generated repOk to Source Code

In Algorithm 2, we first declare all Fields used in the constraint using Variable class(Var) in line 1 and 2. Then, we convert all generated variables to expressions data types in order to apply expression operators on them. Depending on the desired constraint, we relate the different expression using Expression Operators (ExprOperator) to match the desired constraint in line 4 and 5. Similarly to the last step, we utilize the

operations (Op) between generated expressions to build the desired Clauses according to the constraint C. In order to build the final formula, we build the formulas using the generated clauses and we join them using the logical operators. Similar to clauses, we use generated predicates and we join them using the logical operators. Finally, repOk method is generated using f_{final} and append it to the source code. The following example constructs the repOk constraint represented below:

root.elem > root.next.elem && root.next.next == null && root.next != null

The code shown in Figure 9 generates the desired constraints above using the described algorithm:

```
/* ---------------------------- Declare Variables ---------------------------- */
Var elem = new Var("elem","Node", "int");
Var next = new Var("next","Node", "Node");
Var root = new Var("root", "Tree", "Node");
/* ------------------------------ repOk ------------------------------ */
Expr rootExpr = new Expr(root);
Expr elemExpr = new Expr(elem);
Expr nextExpr = new Expr(next);
Expr Expr1 = rootExpr.oper(elemExpr, ExprOperator.Join);
Expr Expr2 = rootExpr.oper(nextExpr, ExprOperator.Join);
Expr Expr3 = Expr2.oper(elemExpr, ExprOperator.Join);
Clause c1 = new Clause(Expr1, Op.BT, Expr3);
Expr Expr4 = Expr2.oper(nextExpr, ExprOperator.Join);
Clause c2 = new Clause(Expr4, Op.EQ, Expr.nullExpr);
Clause c3 = new Clause(Expr2, Op.NEQ, Expr.nullExpr);

Formula f1 = new Formula(c1);
Formula f2 = new Formula(c2);
Formula f3 = new Formula(c3);
Formula f4 = f2.And(f3);
Formula fFinal = f1.And(f4);
```

Figure 9: Set of Calls in Korat-API to generating repOk method for constraint solving problem for standard Korat

The source code generated so far from the field declaration and constraint(*repOk*) generation is demonstrated in Figure 10.

```
public class Tree {
    Node root;
    public class Node {
        Node next;
        int elem;
    }

    public boolean repOK() {
      boolean result = false;
      try {
          result = root.elem > root.next.elem && root.next.next == null && root.next != null;
      } catch (Exception e) {
          ;
      }
      return result;
    }
}
```

Figure 10: Source code generated from field declaration and constraint generation

3.2.3 Generate finitization method

In this part, we demonstrate how we build the finitization method using the finitization data structure introduced in API section. As described earlier, we bound the desired input size by specifying the range of possible values for each field. Similar to how it was described in API section, we follow the next steps in order to translate the set of calls in Korat-API to finitization method for standard Korat:

Algorithm 3: *Add finitization to Source Code*

Input: RangeToField [Range[name, type, args], Set[Var]], finArgs, SourceCode, T_{main}

1. **function** addFinization(RangeToField, finArgs, SourceCode, T_{main})
2. SourceCode.append(InitFin(T_{main}, finArgs)) /* Declare finitization method */
3. **for all** R[name, type, args] ∈ RangeToField.keys() **do**
4. SourceCode.append(R.DeclareRange()) /* Declare range with R */
5. **for all** Var ∈ RangeToField.get(R) **do**
6. SourceCode.append(SetVarRange(Var, R)) /* Set Var field to R */
7. SourceCode.append(EndMethod()) /* add "}" */
8. **return** SourceCode
9. **end function**

Output: Add Finitization Method to Source Code

Algorithm 3 demonstrates how to generate finitization method in source code from constraint solving problem in Korat-API. RangeToField, finArgs, SourceCode, and T_{main} are inputs to this algorithm as shown in line 1. RangeToField is data structure that hold information about all range declarations from our API and what fields are assigned to each range. RangeToField is map between range and set of fields that are assigned to that range. finArgs are finitization arguments that user specifies in order to have the ability to change the range values when invoking Korat. SourceCode is the generated constraint solving problem for standard Korat so far (includes class declarations and repOk method). T_{main} is the name of the main class that contains repOk and finitization methods.

To generate finitization method, we first declare the finitization method using T_{main} and finArgs in line 2. Then, we append it to SourceCode. After that, we iterate over all ranges in RangeToField and add their declaration to SourceCode. Each range has three data members: range name, range type, and range args. For each range, we loop over the set of fields associate with that range. Then, we assign each field with its associated range in source code. Finally, we terminate the finitization method in line 7.

The following example illustrated how the algorithm works. Assume we want to bound the size of fields values used in the previous example. Figure 11 shows how we bound the values of field *elem* to be between *min* and *max* arguments and how we bound the number of possible nodes in field *root* and *next* with argument *num*.

```
/* ---------------------------- Finitization ---------------------------- */
String [] argss = { "int min", "int max" , "int num" };
Fin finitization = new Fin(mainClass, argss);
finitization.CreateIntRange("elemRange", "min", "max");
finitization.SetRange(elem, "elemRange");
finitization.CreateObjRange("nodeRange","Node", true, "num");
finitization.SetRange(next, "nodeRange");
finitization.SetRange(root, "nodeRange");
```

Figure 11: Generating Finitization using Korat-API

If we combine this with the last example, we generate the source code that represent constraint solving problem for standard Korat with all its requirements. The generated source code example is shown in Figure 12.

```
public class Tree {
    Node root;
    public class Node {
        Node next;
        int elem;
    }

    public boolean repOK() {
      boolean result = false;
      try {
          result = root.elem > root.next.elem && root.next.next == null && root.next != null;
      } catch (Exception e) {
          ;
      }
      return result;
    }

    public static IFinitization finNodes(int min, int max, int num) {
      IFinitization f = FinitizationFactory.create(Tree.class);
      IIntSet elemRange = f.createIntSet(min,max);
      f.set("Node.elem", elemRange);
      IObjSet nodeRange = f.createObjSet(Node.class,true);
      nodeRange.addClassDomain(f.createClassDomain(Node.class,num));
      f.set("Node.next", nodeRange);
      f.set("Tree.root", nodeRange);
      return f;
    }
}
```

Figure 12: Example of the generated source code that represent the constraint solving problem for standard Korat

3.2.4 Generate and run constraint solving problem for korat (CSP4K)

There are some more steps are needed in order to generate and run constraint solving problem for Korat. In this section, a combination of all previous three algorithms and the additional steps are shown in Algorithm 4.

Algorithm 4: *Generate and run constraint solving problem for Korat*

Input: Set of Imports (imports), Set of Var API Calls(VC), RangeToField, Set[Var]], finArgs, SourceCode, T_{main}

1. **function** CSP4K (VC , RangeToField, finArgs, SourceCode, T_{main})
2. SourceCode ← Empty String
3. SourceCode.append(AddImports(imports)) /* add imports to start the class */
4. SourceCode.append(DeclareClasses(VC, SourceCode, T_{main})) /* Algorithm 1 */
5. SourceCode.append(GenerateRepOk(f_{final})) /* Algorithm 2 */
6. addFinization(RangeToField, finArgs, SourceCode, T_{main}) /* Algorithm 3 */
7. SourceCode.append(CloseClass()) /* add "}" to end main class */
8. Class cls ← CompilerKorat(SourceCode, T_{main}) /* compile the source code */
9. RunKorat(cls, finArgs) /* Run Korat */
10. **return** 0
11. **end function**

Output: Add Finitization Method to Source Code

As mentioned, Algorithm 4 generates and runs constraint solving problem for Korat. We have covered the algorithms in lines 4, 5 and 6 in the last sections. In line 3, we basically add imports first when we generate the source code. Then, we follow Algorithms 1, 2, and 3 in order to generate class declaration, repOk and finitization respectively. Then, we generate the class in line 8 that contains the constraint solving problem for standard Korat. Finally, Korat runs using the generated class and the desired finitization arguments.

Chapter 4: Demonstration

In the section, we demonstrate additional examples of how to use Korat-API for using Korat as backend constraint solver and standalone test generator. The purpose is to further illustrate how Korat-API allows interfacing with Korat. Also, we believe our API presents an approach where Korat's ability to solve imperative predicates is more efficient and widely applicable.

4.1 SINGLY-LINKED LIST

Consider using Korat to generate sorted singly linked lists with header pointer as test inputs. Figure 13 shows the Java code that defines the constraint solving problem for generating sorted linked lists with header using standard Korat [3]. The class *Entry* introduces the type necessary to model each entry in linked list. Each Linked list has a *header* entry and cashes the number of entries in *size* field. Each *Entry* has an *element Serializable Object*, and *next* for the next *Entry*. *Serializable Object* represents the *element* in *Entry* which keeps track of each entry *ID* and incremented when adding new Entry using static variable *ObjectID*. The predicate *repOk()* uses two helper method to check the desired properties.

The method repOkCommon() checks if the input has a valid singly linked list, i.e. has no (undirected) cycles as well as ensuring the *size* match the number of entries. The method repOkSorted() checks if the *ID* in *element* in the singly linked list is sorted correctly. The finitization method specifies that *size* fields take integer values between

minSize and maxSize (inclusive); and *header* and *next* fields are either null or point to one of numEntries unique *Entry;* and *element* fields are either null or point to one of numElems unique *SerializableObject* that Korat search will create upon initialization.

```
2  import java.util.*;
3  import korat.finitization.*;
4  import korat.finitization.impl.*;
5
6  public class SinglyLinkedList {
7      Entry header;
8      int size;
9      public boolean repOK() {
10         if(!repOkCommon()) return false;
11         if(!repOkSorted()) return false;
12             return true;
13     }
14
15     public boolean repOkSorted() {
16         // Body
17         return true;
18     }
19     private boolean repOkCommon() {
20         // Body
21         return true;
22     }
23
24     public static IFinitization finSinglyLinkedList(int minSize, int maxSize, int numEntries, int numElems) {
25       IFinitization f = FinitizationFactory.create(SinglyLinkedList.class);
26       IIntSet sizes = f.createIntSet(minSize,maxSize);
27       f.set("SinglyLinkedList.size", sizes);
28       IObjSet elems = f.createObjSet(SerializableObject.class,true);
29       elems.addClassDomain(f.createClassDomain(SerializableObject.class,numElems));
30       f.set("Entry.element", elems);
31       IObjSet entries = f.createObjSet(Entry.class,true);
32       entries.addClassDomain(f.createClassDomain(Entry.class,numEntries));
33       f.set("SinglyLinkedList.header", entries);
34       f.set("Entry.next", entries);
35       return f;
36     }
37 }
```

Figure 13: Java code shows the constraint solving problem for generating sorted linked lists using standard Korat[3][http://korat.sourceforge.net/]

To run Korat, the user compiles the Java code that describes solving problem (Figure 4) and invokes Korat using the command line . To illustrate, "*java -cp Korat.jar korat.Korat --class korat.examples.searchtree.LinkedList --args 0,2,2,2"* instructs Korat to enumerate sorted linked list with up to 2 entries and 2 elements where each link list size can be from 0 to 2. Figure 14 shows how Korat-API can be used to define a constraint solving problem that is analogous to the problem represented by Figure 4 and

its corresponding korat invocation above. *Entry* and *SerializableObject* class are both defined in different file (not shown here).

```
45    public static void main(String[] args) {
46        /* ----------------------------- Imports ----------------------------- */
47        String [] imports = {"java.util.*",
48                             "korat.finitization.*",
49                             "korat.finitization.impl.*"};
50        /* ---------------------------- Declare Variables ---------------------------- */
51        Var header = new Var("header","SinglyLinkedList", "Entry");
52        Var size = new Var("size","SinglyLinkedList", "int");
53        /* ------------------------------ repOk ------------------------------ */
54        Predicate repOkCommon = new Predicate("repOkCommon()",repOkCommonMethod);
55        Predicate repOkSorted = new Predicate("repOkSorted()", repOkSortedMethod);
56
57        Formula f1 = repOkCommon.Invocation();
58        Formula f2 = repOkSorted.Invocation();
59        Formula fFinal = f1.And(f2);
60        /* ----------------------------- Finitization ----------------------------- */
61        String mainClass = "SinglyLinkedList";
62        String [] finArgs = { "int minSize, int maxSize, int numEntries, int numElems" };
63        Fin finitization = new Fin(mainClass, finArgs);
64        finitization.CreateIntRange("sizes", "minSize", "maxSize");
65        finitization.SetRange(size, "sizes");
66        finitization.CreateObjRange("elems","SerializableObject", true, "numElems");
67        finitization.SetRange("Entry.element", "elems");
68        finitization.CreateObjRange("entries","Entry", true, "numEntries");
69        finitization.SetRange(header, "entries");
70        finitization.SetRange("Entry.next", "entries");
71
72        /* ----------------------------- Build All ----------------------------- */
73        Builder builder =  Builder.GetBuilder();
74        builder.mainClass(mainClass, fFinal, finitization);
75        builder.build(imports);
76
77        /* ----------------------------- Compile ----------------------------- */
78        CompilerKoratX.compile(builder.getSourceCode(), mainClass);
79
80        /* ----------------------------- Run Korat ----------------------------- */
81        RunKoratx.run(mainClass, "0,2,2,2");
82    }
```

Figure 14: Creating a constraint solving problem for generating sorted linked list using Korat-API

As described in chapter 3, we **first** define fields of the constraint solving problem. Lines 51 and 52 declare the object fields in *SinglyLinkedList* class which are *header* and

size. *header* is of type Entry and under *SinglyLinkedList* class. Therefore, to define header, we use the API call in line 51.

Second, we define the constraint and represented using Korat-API. Since the constraint solving problem for Korat contains helper predicate. We will use the Predicate type introduced in chapter 3. For example, the helper predicate *repOkCommon()* is declared in line 54. We pass the *repOkCommon()* method body as string argument to the declaration method in order to build the predicate. Then, we generate the formula in line 57 based on the created *repOkCommon* predicate. Similarly, we perform the same steps for *repOkSorted()* to generate the formula in line 58.

The *ffinal* is represented with logical conjunction between the formulas generated by the two predicate. Note the invocation of *and()* creates a logical conjunction of two formulas that has to be evaluated in order and not in parallel. From the Java code that describe constraint solving problem in Figure 13, *repOkCommon()* has to be evaluated first in order to evaluate *repOkSorted()* since *repOkCommon()* ensures that linked list is valid in the first place before making sure its sorted.

Third, we define the bounds of state space. For this example, we want to bound the state space (shown in line 81) to have 2 entries and 2 elements where each link list's size is from 0 to 2. In order to bound input size, we start by initializing finitization in line 63 with the main class and its arguments. Then, we define the desired ranges in lines 64, 66, and 68. Since size is integer, its range can be declared using *CreateIntRange* method as shown in line 64. Other fields are object type field, we declared them using *CreateObjRange* method as shown in line 66 and 68. Setting field to range doesn't involve the type of the field. We basically define range for each field by using SetRange(field, RangeName) as shown in lines 65, 67, 69 and 70.

Fourth, we define the analysis to perform. In line 74, we invoke main method that translates the generated constraint solving problem p_a by API-Korat to constraint solving problem p_k for standard Korat. Also, in mainClass method, we specify the type of constraint to solve for and the bound of state space. Line 78 compiles the constraint solving problem p_k for standard Korat and generates the desired class. Lines 81 invokes the default Korat solver to compute the count of the solutions as well as print it (Shown in). *repOkSorted()* and *repOkCommon()* method bodies are shown in Appendix B.

```
Start of Korat Execution for SinglyLinkedList (repOK, [0, 2, 2, 2])

[0]SinglyLinkedList@6ee1dac2.size -> int:3
[1]SinglyLinkedList@6ee1dac2.header -> class Entry:3
[2][null].element -> class SerializableObject:3
[3][null].next -> class Entry:3
[4][null].element -> class SerializableObject:3
[5][null].next -> class Entry:3

0 0 0 0 0 0  :: 1
0 1 0 0 0 0  :: 1 2 3 0
1 1 0 0 0 0  :: 1 2 3 0
2 1 0 0 0 0  :: 1 2 3 0
0 1 0 1 0 0  :: 1 2 3
0 1 0 2 0 0  :: 1 2 3 4
0 1 0 2 1 0  :: 1 2 3 4 5 0
1 1 0 2 1 0  :: 1 2 3 4 5 0 ***
2 1 0 2 1 0  :: 1 2 3 4 5 0
0 1 0 2 1 1  :: 1 2 3 4 5
0 1 0 2 1 2  :: 1 2 3 4 5
0 1 1 0 0 0  :: 1 2
Total explored:12
New found:1
```

Figure 15: The print of the count of the solutions in singly linked list example

4.2 HEAP PATH CONDITIONS FROM SYMBOLIC PATHFINDER

In the following example, we will demonstrate how our API is able to provide the necessary support to construct constraint solving problem when predicate is based on path conditions from symbolic execution, e.g., using Symbolic PathFinder [24]. In Figure 16, the Java code for NodeSimple class is presented which contains only two members: integer *elem* and SimpleNode *next*. This class has very simple structure with only three methods which are *test()*, *NodeSimple()*, and *main()*. Using symbolic execution on method test(), we extracted the five path conditions below:

1. *root != null && root.next != null && root.next != root*
2. *root.next == null && root != null*
3. *root.next == root && root != null*
4. *root != null*
5. *root == null*

For each path condition, we show to construct a constraint solving problem using Korat-API; we term these problems SimpleNode1, SimpleNode2, …, SimpleNode5.

```java
package symbolicheap;

import gov.nasa.jpf.symbc.Debug;

public class NodeSimple {

    int elem;
    NodeSimple next;


    public NodeSimple(int x) {
        elem = x;
        next = null;
    }

    public void test(NodeSimple n) {
        if(n!=null && n.next!=null)
            System.out.println("2 elements");
    }
    public static void main(String[] args) {

        NodeSimple X = new NodeSimple(5);
        (new NodeSimple(0)).test(X);
    }

}
```

Figure 16: Java code for NodeSimple Class from standard examples for Symbolic PathFinder [24][http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc]

```
1  import Design.*;
2
3  public class RunSimpleNode1 {
4      public static void main(String[] args) {
5          String mainClass = "SimpleNode1";
6          /* ---------------------------- Imports ---------------------------- */
7          String [] imports = {"java.util.*",
8                               "korat.finitization.*",
9                               "korat.finitization.impl.*"};
10
11         /* ---------------------------- Declare Variables ---------------------------- */
12         Var elem = new Var("elem","NodeSimple", "int");
13         Var next = new Var("next","NodeSimple", "NodeSimple");
14         Var root = new Var("root", "SimpleNode1", "NodeSimple");
15
16         /* ---------------------------- repOk ---------------------------- */
17         Expr rootExpr = new Expr(root);
18         Expr nextExpr = new Expr(next);
19         Expr Expr1 = rootExpr.oper(nextExpr, ExprOperator.Join);
20         Clause c1 = new Clause(rootExpr, Op.NEQ, Expr.nullExpr);
21         Clause c2 = new Clause(Expr1, Op.NEQ, Expr.nullExpr);
22         Clause c3 = new Clause(Expr1, Op.NEQ, rootExpr);
23         Formula f1 = new Formula(c1);
24         Formula f2 = new Formula(c2);
25         Formula f3 = new Formula(c3);
26         Formula f4 = f2.lAnd(f3);
27         Formula fFinal = f1.And(f4);
28         /* ---------------------------- Finitization ---------------------------- */
29         String [] argss = {"int min", "int max", "int num"};
30         Fin finitization = new Fin(mainClass, argss);
31         finitization.CreateIntRange("elems", "min", "max");
32         finitization.SetRange(elem, "elems");
33         finitization.CreateObjRange("nodes","NodeSimple", true, "num");
34         finitization.SetRange(next, "nodes");
35         finitization.SetRange(root, "nodes");
36         /* ---------------------------- Build All ---------------------------- */
37         Builder builder =  Builder.GetBuilder();
38         builder.mainClass(mainClass, fFinal, finitization);
39         builder.build(imports);
40         /* ---------------------------- Compile ---------------------------- */
41         CompilerKoratX.compile(builder.getSourceCode(), mainClass);
42         /* ---------------------------- Run Korat ---------------------------- */
43         RunKoratx.run(mainClass, "0,1,2");
44     }
45 }
```

Figure 17: Creating a constraint solving problem for counting the number of solutions for path condition " root != null && root.next != null && root.next != root "

Using our API, we want to construct the constraint solving problem for path condition 1 above. Using the API and algorithm, we generated constraint solving problem, in Figure 17, that is analogous to constraint problem for the path condition "*root != null && root.next != null && root.next != root*". We have generated the constraint solving problem for all five path conditions and we named them from SimpleNode1 to

SimpleNode5. For sake of simplicity, we will go over the construction of SimpleNode1 only and show the results of the rest of them.

In SimpleNode1 constraint solving problem, Lines 7-9 list the imports needed to run the problem. Lines 12-14 declare the object fields. Lines 17-27 define the *repOk()* predicate. Lines 31 and 33 define the overall value domain. Lines 32, 34, and 35 define for each field, its value domain. Lines 37-43 build the constraint solving problem and invoke the default Korat solver to compute the count of the solutions as well as print it.

We build the constraint solving problem for the rest of path condition in the same way following the API and algorithm introduced in chapter 3. Figure 18 to Figure 22 represents the count of the solutions for SimpleNode1 to SimpleNode5 respectively. The only different on generating the constraint solving problem for those five path conditions is the predicate. The field definition and bound of state remain the same for all constructing the constraint solving problem for those five path conditions.

```
Start of Korat Execution for List (repOK, [0, 3, 3])

[0]List@c2f8b5a.size -> int:4
[1]List@c2f8b5a.header -> class List$Entry:4
[2]List$Entry@565c7f6.next -> class List$Entry:4
[3]List$Entry@5d2aea3e.next -> class List$Entry:4
[4]List$Entry@2c78bc3b.next -> class List$Entry:4

0 0 0 0 0  :: 1
0 1 0 0 0  :: 1 2
0 1 1 0 0  :: 1 2 0
1 1 1 0 0  :: 1 2 0
2 1 1 0 0  :: 1 2 0 ***
3 1 1 0 0  :: 1 2 0
0 1 2 0 0  :: 1 2 3
0 1 2 1 0  :: 1 2 3 0
1 1 2 1 0  :: 1 2 3 0
2 1 2 1 0  :: 1 2 3 0 ***
3 1 2 1 0  :: 1 2 3 0
0 1 2 2 0  :: 1 2 3
0 1 2 3 0  :: 1 2 3
Total explored:13
New found:2
```

Figure 18: Korat output for counting solutions for SimpleNode1

```
Start of Korat Execution for SimpleNode2 (repOK, [0, 1, 2])

[0]SimpleNode2@21e1962d.root -> class SimpleNode2$NodeSimple:3
[1]SimpleNode2$NodeSimple@614a75bb.elem -> int:2
[2]SimpleNode2$NodeSimple@614a75bb.next -> class SimpleNode2$NodeSimple:3
[3]SimpleNode2$NodeSimple@5f7779e3.elem -> int:2
[4]SimpleNode2$NodeSimple@5f7779e3.next -> class SimpleNode2$NodeSimple:3

0 0 0 0 0  :: 0
1 0 0 0 0  :: 0 2 ***
1 1 0 0 0  :: 0 2 ***
1 0 1 0 0  :: 0 2
1 0 2 0 0  :: 0 2
Total explored:5
New found:2
```

Figure 19: Korat output for counting solutions for SimpleNode2

```
Start of Korat Execution for SimpleNode3 (repOK, [0, 1, 2])

[0]SimpleNode3@7bd6747b.root -> class SimpleNode3$NodeSimple:3
[1]SimpleNode3$NodeSimple@3c7976c1.elem -> int:2
[2]SimpleNode3$NodeSimple@3c7976c1.next -> class SimpleNode3$NodeSimple:3
[3]SimpleNode3$NodeSimple@4a6397eb.elem -> int:2
[4]SimpleNode3$NodeSimple@4a6397eb.next -> class SimpleNode3$NodeSimple:3

0 0 0 0 0  :: 0
1 0 0 0 0  :: 0 2
1 0 1 0 0  :: 0 2 ***
1 1 1 0 0  :: 0 2 ***
1 0 2 0 0  :: 0 2
Total explored:5
New found:2
```

Figure 20: Korat output for counting solutions for SimpleNode3

```
Start of Korat Execution for SimpleNode4 (repOK, [0, 1, 2])

[0]SimpleNode4@7df44ec7.root -> class SimpleNode4$NodeSimple:3
[1]SimpleNode4$NodeSimple@32602b6b.elem -> int:2
[2]SimpleNode4$NodeSimple@32602b6b.next -> class SimpleNode4$NodeSimple:3
[3]SimpleNode4$NodeSimple@47c297a3.elem -> int:2
[4]SimpleNode4$NodeSimple@47c297a3.next -> class SimpleNode4$NodeSimple:3

0 0 0 0 0  :: 0
1 0 0 0 0  :: 0 ***
1 0 1 0 0  :: 0 ***
1 0 2 0 0  :: 0 ***
1 0 2 0 1  :: 0 ***
1 0 2 0 2  :: 0 ***
1 0 2 1 0  :: 0 ***
1 0 2 1 1  :: 0 ***
1 0 2 1 2  :: 0 ***
1 1 0 0 0  :: 0 ***
1 1 1 0 0  :: 0 ***
1 1 2 0 0  :: 0 ***
1 1 2 0 1  :: 0 ***
1 1 2 0 2  :: 0 ***
1 1 2 1 0  :: 0 ***
1 1 2 1 1  :: 0 ***
1 1 2 1 2  :: 0 ***
Total explored:17
New found:16
```

Figure 21: Korat output for counting solutions for SimpleNode4

```
Start of Korat Execution for SimpleNode5 (repOK, [0, 1, 2])

[0]SimpleNode5@2a313170.root -> class SimpleNode5$NodeSimple:3
[1]SimpleNode5$NodeSimple@3a4c5b4.elem -> int:2
[2]SimpleNode5$NodeSimple@3a4c5b4.next -> class SimpleNode5$NodeSimple:3
[3]SimpleNode5$NodeSimple@36afae4a.elem -> int:2
[4]SimpleNode5$NodeSimple@36afae4a.next -> class SimpleNode5$NodeSimple:3

0 0 0 0 0  :: 0 ***
1 0 0 0 0  :: 0
Total explored:2
New found:1
```

Figure 22: Korat output for counting solutions for SimpleNode5

4.3 BINARY SEARCH TREE

Consider using Korat to generate binary search tree with parent pointers as test inputs. Figure 23 shows the Java code that defines the constraint solving problem for generating binary search trees with parent pointers using standard Korat [3]. The class Node introduce the type necessary to model the tree structure. Each tree has a *root* node and cashes the number of nodes in *size* field. Each node has a *left* child node, a *right* child node , and integer value *info*. The predicate *repOk()* uses four helper method (Omitted in Figure 23, but See Appendix A) to check the desired properties.

The method checkRootSize() checks if root is null, then size has to be zero. The method isAcyclic() checks if the input has a valid tree structure, i.e. has no (undirected) cycles. The method sizeOk() checks that all nodes have the correct value in size field starting from the root node. The method isOrdered() checks if the *info* value in the tree nodes trees are in the correct sorted order (Inorder). The finitization method specifies that *info* fields take integer values between minData and maxData (inclusive) and *size* take integer values between minSize and maxSize (inclusive); and *root*, *left*, and *right* fields are either null or point to one of numNodes unique nodes that Korat search will create upon initialization.

```
public class SearchTree {
    Node root;
    int size;

    public boolean repOK() {

      if(!checkRootAndSize()) return false;

       if(!isAcyclic()) return false;

       if(!sizeOk()) return false;

       if(!isOrdered(root,-1,-1)) return false;

       return true;
    }

    // Method declarations here ...

    public static IFinitization finSearchTree(int numNodes, int minSize, int maxSize, int minData, int maxData) {
          IFinitization f = FinitizationFactory.create(SearchTree.class);
          IIntSet sizes = f.createIntSet(minSize,maxSize);
          f.set("SearchTree.size", sizes);
          IIntSet values = f.createIntSet(minData,maxData);
          f.set("Node.info", values);
          IObjSet nodes = f.createObjSet(Node.class,true);
          nodes.addClassDomain(f.createClassDomain(Node.class,numNodes));
          f.set("SearchTree.root", nodes);
          f.set("Node.left", nodes);
          f.set("Node.right", nodes);
          return f;
    }
}
```

Figure 23: Java code show the constraint solving problem for generating search trees using standard Korat [3][http://korat.sourceforge.net/]

To run Korat, the user compiles the Java code that describes solving problem (Figure 23) and invokes Korat using the command line . To illustrate, *"java -cp Korat.jar korat.Korat --class korat.examples.searchtree.SearchTree --args 2,1,2,1,2"* instructs Korat to enumerate all trees with up to 2 nodes where each node element is 1 or 2. Figure 24 shows how Korat-API can be used to define a constraint solving problem that is analogous to the problem represented by Figure 4 and its corresponding korat invocation above. Node class is defined in different file (not shown here). Lines 65-67 list the imports needed to run the problem. Lines 69 and 70 declare the object fields. Lines 72-83 define the *repOk()* predicate. Note the invocation of *land()* creates a logical conjunction of two formulas that can be evaluated in any order. Lines 88,90 and 92

define the overall value domain. Lines 89, 91,93, 94 and 95 define for each field, its value domain. Lines 98-106 build the constraint solving problem and invoke the default Korat solver to compute the count of the solutions as well as print it. The four helper methods were provided as string to the predicate call.

```
63      public static void main(String[] args) {
64          /* ----------------------------- Imports ----------------------------- */
65          String [] imports = {"java.util.*",
66                               "korat.finitization.*",
67                               "korat.finitization.impl.*"};
68          /* ---------------------------- Declare Variables ---------------------------- */
69          Var root = new Var("root","SearchTree", "Node");
70          Var size = new Var("size","SearchTree", "int");
71          /* ------------------------------ repOk ------------------------------ */
72          Predicate checkRootSize = new Predicate("checkRootAndSize()", checkRootAndSizeMethod);
73          Predicate isAcyclic = new Predicate("isAcyclic()",isAcyclicMethod);
74          Predicate sizeOk = new Predicate("sizeOk()", sizeOkMethod);
75          Predicate isOrdered = new Predicate("isOrdered(root,-1,-1)", isOrderedMethod);
76
77          Formula f1 = checkRootSize.Invocation();
78          Formula f2 = isAcyclic.Invocation();
79          Formula f3 = sizeOk.Invocation();
80          Formula f4 = isOrdered.Invocation();
81          Formula f5 = f1.And(f2);
82          Formula f6 = f3.lAnd(f4);
83          Formula fFinal = f5.And(f6);
84          /* ---------------------------- Finitization ---------------------------- */
85          String mainClass = "SearchTree";
86          String [] finArgs = { "int numNodes, int minSize, int maxSize, int minData, int maxData" };
87          Fin finitization = new Fin(mainClass, finArgs);
88          finitization.CreateIntRange("sizes", "minSize", "maxSize");
89          finitization.SetRange(size, "sizes");
90          finitization.CreateIntRange("values", "minData", "maxData");
91          finitization.SetRange("Node.info", "values");
92          finitization.CreateObjRange("nodes","Node", true, "numNodes");
93          finitization.SetRange(root, "nodes");
94          finitization.SetRange("Node.left", "nodes");
95          finitization.SetRange("Node.right", "nodes");
96
97          /* ----------------------------- Build All ----------------------------- */
98          Builder builder =  Builder.GetBuilder();
99          builder.mainClass(mainClass, fFinal, finitization);
100         builder.build(imports);
101
102         /* ----------------------------- Compile ----------------------------- */
103         CompilerKoratX.compile(builder.getSourceCode(), mainClass);
104
105         /* ----------------------------- Run Korat ----------------------------- */
106         RunKoratx.run(mainClass, "2,1,2,1,2");
107     }
108 }
```

Figure 24: Creating a constraint solving problem for generating search trees using Korat-API

Chapter 5: Discussion

Similar to using other constraint solvers [2], [3], [19], there are some potential issues that may arise when using the Korat-API. This chapter discusses some of these issues, which the user may want to pay particular attention to.

While Korat-API introduces a new way to use Korat as backend server and as a test generator, the user of Korat-API requires using a different way to write imperative predicates than it is common for writing Java methods. Specifically, if the user manually writes the *repOk()* predicate using Korat-API, the user has to take specific care to conform to Java language syntax and semantics. Using standard Korat, the user can simply leverage a standard IDE, such as Eclipse, which provides incremental compilation and can highlight any compilation error as the user writes *repOk()*. In contrast, when using Korat-API, Java compilation errors in the repOk() body may be detected much later, and may be harder to debug. However, this issue doesn't arise when Korat-API is used programmatically by a tool to create constraint solving problems.

Korat-API user also need to take special care when defining logical conjunction and separable constraints. If the user erroneously defines a conjunction to be logical and the backend solver tries to exploit it by constraint re-ordering, an exception may occur. Similarly, if the user incorrectly defines a conjunction over separable constraints when they have a common set of fields, e.g., they each constrain *left* and *right* fields, and the backend solver tries to exploit it, a contradiction may occur.

Chapter 6: Related Work

This section describes the related work. We first discuss the Kodkod [19] backend of Alloy [20]. Next, we discuss some projects that build on the basic Korat framework [3] to enhance it.

Our work take inspiration from the success of Kodkod, which has been used in a number of applications [21]. Kodkod provides an API to build Alloy models programmatically and to solve them using off-the-shelf propositional satisfiability (SAT) solvers. The Kodkod API allows declaring the basic relations in the model, writing Alloy formulas over the relations, invoking Alloy commands that direct SAT solving, and reporting the solutions. Kodkod also implements a number of optimizations to provide efficient analysis for Alloy. We believe our Korat-API can leverage further insights into the design and implementation of Kodkod and make Korat even more useful. A key difference however between Korat and SAT is that Korat solves imperative predicates which have a more complex structure and semantics than propositional formulas that SAT solves. However, we believe the basic ideas at the heart of Kodkod can allow future work to create optimized problems that Korat can handle more efficiently.

Our work is also motivated by recent work on software reliability that uses Korat as a backend model counter [11], [12]. While the goal of these projects, i.e., to use Korat – as is – in a specific problem context, was quite different from our goal of designing Korat-API to directly facilitate such use of Korat, they provided us additional motivation to design Korat-API. Parallel Korat [13] introduced the first parallel technique for test generation and execution in the context of Korat. A key contribution of this project was

to introduce the idea of ranging where Korat is run to explore a part of the input space that is defined by a given range. Follow-up work on PKorat [14] introduced a work-list based algorithm for distributing the Korat search among different workers and used work stealing for load balancing. More recent work [15] introduced infeasible ranges that characterize parts of space that Korat explores without finding any valid input to optimize parallel Korat. Our Korat-API introduces a new way to support parallel analysis, e.g., by solving separable constraints in parallel and combining the solutions.

Incremental techniques for Korat were introduced most recently [15] to memorize key steps of Korat search in solving one constraint solving problem and re-use them when possible when solving the next problem. Our Korat-API introduces a foundation that can support in future work incremental analysis directly by providing appropriate constructs, e.g., to incrementally define constraints and finitization bounds, thereby bringing the spirit of incremental SAT to Alloy.

Static analysis [16] and dynamic analysis [17], [22], [23], have also been used to enhance Korat. The goal of these analyses is to guide the Korat search to reduce the amount of exploration. Our Korat-API provides a way to directly guide some of these analyses and more directly improve the Korat search's effectiveness

Chapter 7: Conclusion

This thesis introduced the foundation of an API for building constraint solving problems for the Korat solver for imperative predicates. Our goal was two-fold: (1) to facilitate the use of Korat as a backend solver for applications that desire using it as a constraint solving engine; and (2) to facilitate optimized analyses using Korat, which follow the spirit of modern constraint solving and software testing techniques. We described the API and how it uses the core Korat engine, and demonstrated the benefits in two application contexts: (1) using Korat as a backend engine for model counting; and (2) using Korat as a test generator. We believe our work introduces a promising approach for making the ability of Korat to efficiently solve imperative predicates more widely applicable, possibly even in new application contexts where Korat has not been used before

Appendices

APPENDIX A

The following figure shows the body for methods *checkRootAndSize*, *SizOk*, *isAcyclic* and *isOrdered* based on standard Korat examples [3][http://korat.sourceforge.net/].

```
private boolean checkRootAndSize() {
    if (root == null)
        return size == 0;
    return true;
}

private boolean sizeOk() {
    if (root == null) {
        if (0 != size)
            return false;
    } else {
        if (1 + numNodes(root.left) + numNodes(root.right) != size)
            return false;
    }
    return true;
}

private boolean isAcyclic() {
    Set visited = new HashSet();
    visited.add(root);
    LinkedList workList = new LinkedList();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = (Node) workList.removeFirst();
        if (current.left != null) {
            // checks that the tree has no cycle
            if (!visited.add(current.left))
                return false;
            workList.add(current.left);
        }
        if (current.right != null) {
            // checks that the tree has no cycle
            if (!visited.add(current.right))
                return false;
            workList.add(current.right);
        }
    }
    return true;
}
```

```
private boolean isOrdered(Node n, int min, int max) {
    // if (n.info == null)
    // return false;
    if (n.info == -1)
        return false;
    // if ((min != null && n.info.compareTo(min) <= 0)
    // || (max != null && n.info.compareTo(max) >= 0))
    if ((min != -1 && n.info <= (min)) || (max != -1 && n.info >= (max)))

        return false;
    if (n.left != null)
        if (!isOrdered(n.left, min, n.info))
            return false;
    if (n.right != null)
        if (!isOrdered(n.right, n.info, max))
            return false;
    return true;
}
```

APPENDIX B

The following figure shows the body for methods *repOkCommon* and *repOkSorted* based on standard Korat examples [3][http://korat.sourceforge.net/]

```
public boolean repOkSorted() {
    if (header == null || header.next == null)
        return false;
    if ((header.next == header) && (!(header.next.element instanceof Comparable)))
        return false;
    for (Entry current = header.next; current.next != null; current = current.next) {
        if ((!(current.next.element instanceof Comparable))
                || (((Comparable) current.element).compareTo((Comparable) current.next.element) > 0))
            return false;
    }
    return true;
}
private boolean repOkCommon() {
    if (header == null)
        return false;
    if (header.element != null)
        return false;
    Set<Entry> visited = new java.util.HashSet<Entry>();
    visited.add(header);
    Entry current = header;
    while (true) {
        Entry next = current.next;
        if (next == null)
            break;
        if (next.element == null)
            return false;
        if (!visited.add(next))
            return false;
        current = next;
    }
    if (visited.size() - 1 != size)
        return false;
    return true;
}
```

References

[1] N. Een and N. Sorensson, “An extensible SAT-solver,” in SAT, Santa Margherita Ligure, Italy, 2003.

[2] L. de Moura and N. Bjorner, “Z3: An efficient SMT solver,” in TACAS, Budapest, Hungary, 2008.

[3] C. Boyapati, S. Khurshid, and D. Marinov, “Korat: Automated testing based on Java predicates,” in International Symposium on Software Testing and Analysis. New York, NY, USA: ACM, 2002, pp. 123–133.

[4] J. C. King, “Symbolic execution and program testing,” CACM, vol. 19, no. 7, 1976.

[5] S. Khurshid, C. Pasareanu, and W. Visser, “Generalized symbolic execution for model checking and testing,” in TACAS, April 2003.

[6] P. Godefroid, N. Klarlund, and K. Sen, “DART: directed automated random testing,” in PLDI, 2005.

[7] D. Marinov and S. Khurshid, “TestEra: A novel framework for automated testing of Java programs,” in ASE, Nov. 2001, pp. 22–31.

[8] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, “The lean theorem prover,” in CADE-25, 2015.

[9] B. Liskov and J. Guttag, Program Development in Java: Abstraction, Specification, and Object-Oriented Design, 2000.

[10] D. Marinov, “Automatic testing of software with structurally complex inputs,” Ph.D. dissertation, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004.

[11] A. Filieri, M. F. Frias, C. S. Pasareanu, and W. Visser, “Model counting for complex data structures,” in SPIN, 2015, pp. 222–241.

[12] A. Filieri, C. S. Pasareanu, and W. Visser, “Reliability analysis in symbolic pathfinder,” in ICSE, 2013, pp. 622–631.

[13] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov, “Parallel test generation and execution with Korat,” in International Symposium on Foundations of Software Engineering. New York, NY, USA: ACM, Sep 2007, pp. 135–144.

[14] J. H. Siddiqui and S. Khurshid, “PKorat: Parallel generation of structurally complex test inputs,” in International Conference on Software Testing Verification and Validation. Denver, CO, USA: IEEE, Apr 2009, pp. 250–259.

[15] N. Dini, “MKorat: A novel approach for memoizing the Korat search and some potential applications,” Master’s thesis, University of Texas at Austin, 2016.

[16] R. Srinivasan, “Improving constraint-based test input generation using Korat,” Master’s thesis, University of Texas at Austin, 2015.

[17] A. V. Kulkarni, “Constraint prioritization for efficient test generation using Korat,” Masters report, University of Texas at Austin. 2007.

[18] P. Ammann and J. Offutt, Introduction to Software Testing. Cambridge University Press, 2007.

[19] E. Torlak and D. Jackson, “Kodkod: A relational model finder,” in TACAS, 2007, pp. 632–647.

[20] D. Jackson, Software Abstractions: Logic, Language, and Analysis. Cambridge, MA: The MIT Press, 2006.

[21] Kodkod applications website, “http://emina.github.io/kodkod/apps.html.”

[22] B. Elkarablieh, D. Marinov, and S. Khurshid, “Efficient solving of structural constraints,” in ISSTA, 2008, pp. 39–50.

[23] J. H. Siddiqui, D. Marinov, and S. Khurshid, “Optimizing a structural constraint solver for efficient software checking,” in ASE, 2009, pp. 615–619.

[24] Corina S. Pasareanu, Neha Rungta. “Symbolic PathFinder: symbolic execution of Java bytecode,” ASE 2010

Vita

Zakaria Alrmaih was born in Tripoli, Libya. He graduated from Northeastern University with Bachelor of science in Electrical Engineering. He is pursuing graduate studies in University of Texas at Austin.

Email: zakotm@yahoo.com

This thesis was typed by Zakaria Alrmaih.